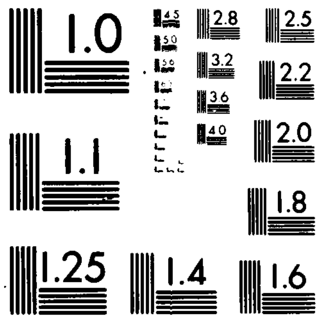

MICROCOPY RESOLUTION TEST CHART

12

RADC-TR-80-138 Vol II (of three)
Final Technical Report
April 1980

SELF-METRIC SOFTWARE
A Handbook: Part I, Logical Ripple Effect Analysis

Northwestern University

Stephen S. Yau
James S. Collofello
Chung-Chu Hsieh

400351

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-80-138, Vol II (of three) has been reviewed and is approved for publication.

APPROVED: Rocco F. Iuorno

ROCCO F. IUORNO
Project Engineer

APPROVED: [illegible]

WENDALL C. BAUMAN, Col, USAF
Chief, Information Sciences Division

Accession For	
NTIS GRA&I	✓
DDC TAB	
Unannounced	
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or special
A	

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS), Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

19 TR-8Ø-138-Vol-2

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
~~1. REPORT NUMBER~~ 18 RADC-TR-80-138, Vol II (of three)	2. GOVT ACCESSION NO. AD-A086 291	3. RECIPIENT'S CATALOG NUMBER
~~4. TITLE (and Subtitle)~~ 6 SELF-METRIC SOFTWARE. Volume II. A Handbook. Part I. Logical Ripple Effect Analysis.		~~5. TYPE OF REPORT & PERIOD COVERED~~ 9 Final Technical Report Aug 76 — Jan 80
		~~6. PERFORMING ORG. REPORT NUMBER~~ N/A
~~7. AUTHOR(s)~~ 10 Stephen S. Yau, James S. Collofello, Chung-Chu Hsieh		8. CONTRACT OR GRANT NUMBER(s) 15 F30602-76-C-0397
~~9. PERFORMING ORGANIZATION~~ NAME AND ADDRESS Northwestern University, Department of Electrical Engineering & Computer Science Evanston IL 60201		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 16 55810278 17 Ø2
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441		~~12. REPORT DATE~~ 11 April 1980
		13. NUMBER OF PAGES 31
14. MONITORING ~~AGENCY NAME &~~ ADDRESS (if different from Controlling Office) Same 12 4Ø		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Rocco F. Iuorno (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software maintenance process, logical ripple effect analysis, technique, handbook, lexical analysis and tracing, block and module error characteristics, intramodule and intermodule error flows.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This handbook consists of two parts on ripple effect analysis for large-scale software maintenance. In Part I, a ripple effect analysis technique for software maintenance from the logical or functional perspective is presented. In a separate volume, the Part II of the handbook, a ripple effect analysis technique for software maintenance from the performance perspective is presented. The purpose of this handbook is to present ripple effect analysis techniques to assist software maintenance person-

(Cont'd)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Item 20 (Cont'd)

nel to do a better job in large-scale software maintenance. The material presented in this handbook is organized in three levels. At the first level, the software maintenance process is described and the need for effective ripple effect analysis techniques for large-scale software maintenance is given. The capabilities and restrictions of the logical ripple effect analysis technique, as well as how this technique is interfaced with the user, are presented. At the second level, the logical ripple effect analysis technique is outlined in two phases: the lexical analysis phase and the tracing phase. At the third level, the steps of the logical ripple effect analysis technique are given in detail. However, the detailed theory behind this technique is not presented in this handbook, but contained in other reports.

Table of Contents

List of Figures

1.0 Introduction

The amount of the maintenance effort in the life cycle of large-scale software has been large and continuously increasing. Software maintenance is a very broad activity that includes error corrections, enhancements of capabilities, deletion of obsolete capabilities, and minor modifications in mission requirements [1-6]. Optimization is also a form of maintenance requiring the modification of code within individual modules, and possibly the structure of the complete system in order to improve its efficiency [7]. The software life cycle has been analyzed to determine the relative magnitude of this maintenance activity with respect to other software development phases [8,9]. The results of one study [9] are illustrated in Figure 1, and indicate that the cost of maintenance is 67% of the total cost for the life cycle of large-scale software. It is obvious that in order to reduce the high cost of software, the most effective approach is to understand the nature of software maintenance and develop effective maintenance techniques. This requires a clear understanding of what is meant by the maintainability of a software system.

The maintainability of a software system can be defined as a measure of the ease of making modifications to the software. In software, the effect of a modification may not be local to the location of the modification, but may also affect other portions of the system. There is a ripple effect from the location of the modification to the other parts of the system that are affected by the modification [1,10-12]. One aspect of this ripple effect is logical in nature. It involves identifying program areas which require additional maintenance activity to insure that consistency with the initial change. Another aspect of this ripple effect concerns the performance of the system. It involves analyzing changes to one program area which may affect the performance of other program areas. Ripple effect analysis techniques are needed for analyzing this ripple effect from both a logical and a performance perspective. This is required since a large-scale program usually has both functional and performance requirements. The ripple effect analysis techniques needed are put into perspective within the maintenance process in Figure 2. As illustrated, the techniques are applied after the maintenance personnel have generated one or more maintenance proposals.

This handbook is divided into two parts with each part describing one of the ripple effect analysis techniques. Part I describes the logical ripple

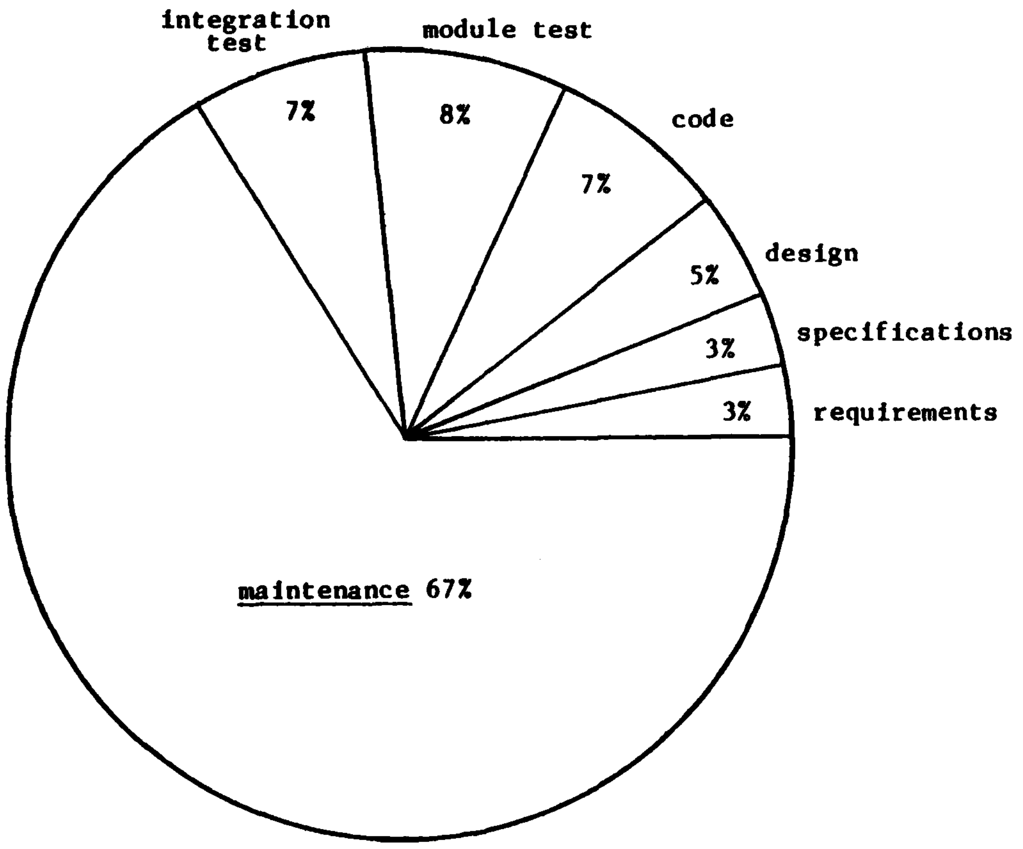

Figure 1. A recent estimate of the life cycle cost for large-scale software [9]

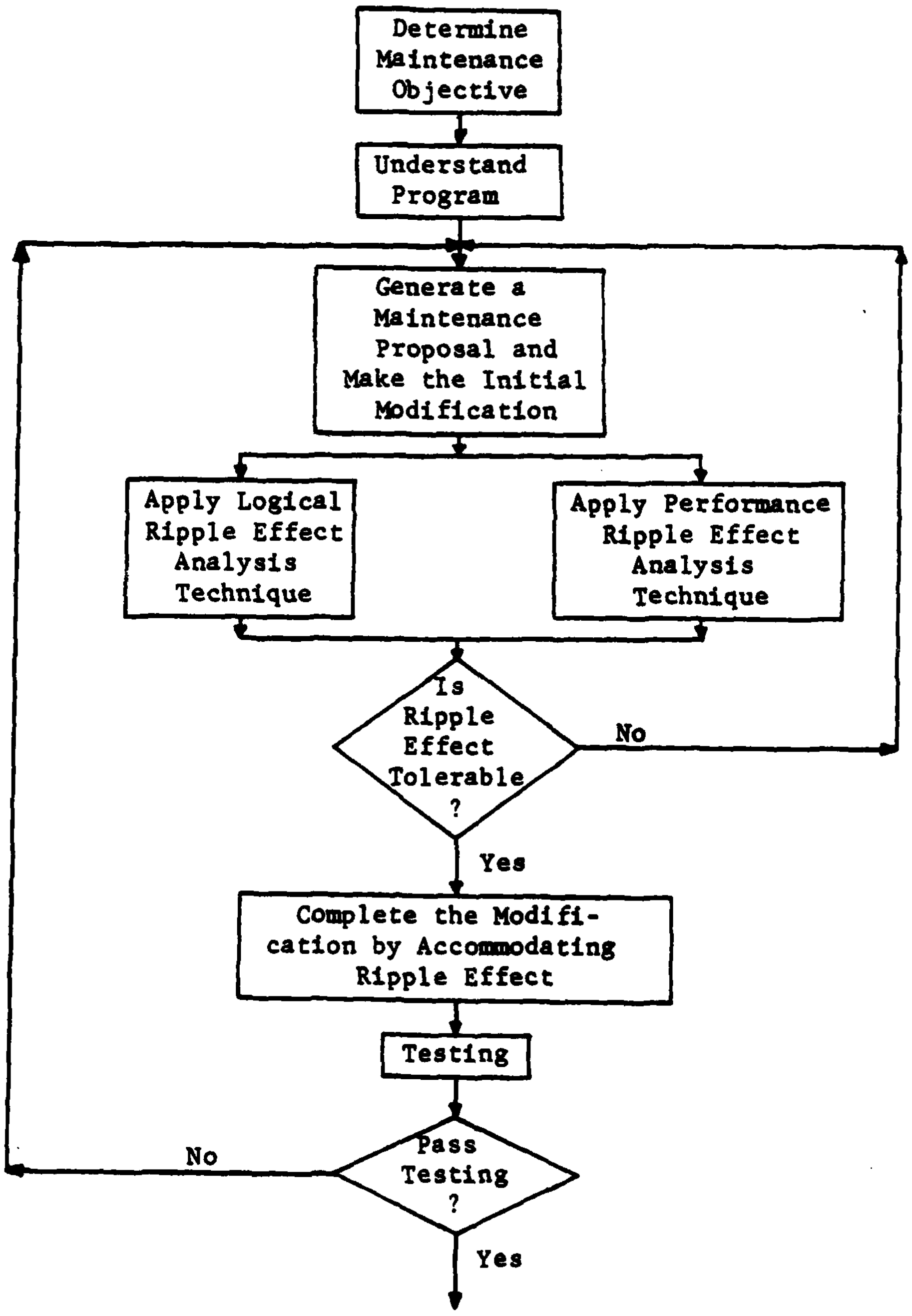
Determine Maintenance Objective
Understand Program
Generate a Maintenance Proposal and Make the Initial Modification
Apply Logical Ripple Effect Analysis Technique
Apply Performance Ripple Effect Analysis Technique
Is Ripple Effect Tolerable ?
No
Yes
Complete the Modification by Accommodating Ripple Effect
Testing
Pass Testing ?
No
Yes

Figure 2. A software maintenance process with the ripple effect analysis techniques

effect analysis technique. Part II describes the performance ripple effect analysis technique [13]. Part I is completely independent of Part II since logical ripple effect analysis can be applied without performance ripple effect analysis. Part II references Part I since performance ripple effect analysis should be utilized in conjunction with logical ripple effect analysis.

The logical ripple effect analysis technique presented in this part of the handbook can be a powerful tool for maintenance personnel. Figure 3 expands the logical ripple effect analysis box in Figure 2 to illustrate the inputs and outputs of the technique. The outputs of the technique can help maintenance personnel understand the scope of effect of their changes on the program. They can also aid them in determining which parts of the program must be checked for consistency. The net results of applying the logical ripple effect analysis technique are:

* Smoother implementation of program modifications

* Reduction of program errors introduced in the program due to modifications

* Reduction of program structure degradation as a consequence of program modification due to an increased understanding of the implications of the modification

* Decrease of the growth rate of program complexity due to program modification

* Extension of overall program's operating life

Another significant product of the logical ripple effect analysis technique is the computation of the complexity of a proposed program modification. One such figure has been proposed which reflects the amount of work involved in performing maintenance and, thus, provides a standard on which comparisons of modifications can be made [1]. However, further research is required for estimating such a figure.

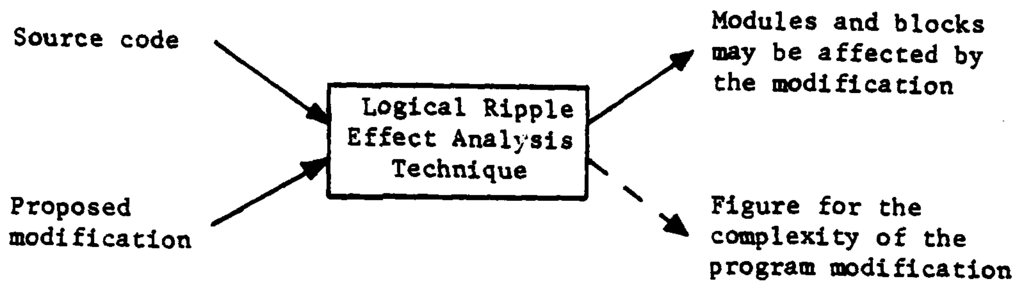
Source code
Proposed modification
Logical Ripple Effect Analysis Technique
Modules and blocks may be affected by the modification
Figure for the complexity of the program modification

Figure 3. Illustration of the inputs and outputs of the logical ripple effect analysis technique

The objective of Part I of this handbook is to describe the logical ripple effect analysis technique in a clear and concise manner. Section 2 describes the capabilities and restrictions of the technique. Section 3 presents the user level interface to this technique as it is perceived to be when fully implemented. Section 4 outlines the required processing necessary to accomplish the functions described in Section 3. The remainder of the sections deal with a description of each of the processing steps. This handbook does not contain implementation details or verification of the algorithms described. Detailed information of this type is discussed in other reports [12,14,15].

2.0 Capabilities and Restrictions of the Logical Ripple Effect Analysis Technique

The logical ripple effect analysis technique described in this part of the handbook is language independent and applicable to existing programs as well as newly implemented programs incorporating state-of-the-art design techniques. The technique does not provide maintenance personnel with proposals for modifying the program. Instead, the technique is applied after the maintenance personnel have generated a number of possible maintenance proposals.

The current version of the logical ripple effect analysis technique also makes the following assumptions about the program to be analyzed.

1. The program does not contain any recursive procedures.

In the current version of the logical ripple effect analysis technique, it is assumed that the module invocation graph is acyclic. Hence, recursion is not allowed. Programming languages such as JOVIAL, FORTRAN, COBOL, do not have recursion. Nevertheless, we will eliminate this restriction in our future research.

2. Statement names and module names cannot be passed between modules.

A module is defined to be a separately invokable piece of the software system having single entry and single exit points. If statement names and module names can be passed between modules, then the invocated module may have multiple entry or multiple exit points. However, current programming practices try to avoid this feature because it leads to bad programming style. Thus, it can be justified that statement names and module names should not be passed between modules.

Another limitation of the current version of the logical ripple effect analysis technique is that the technique is oriented towards tracing data flow. Thus, the technique provides limited information concerning logical ripple effect if the initial modification changes only the control flow of the program. Future research will be focused on covering the ripple effect analysis due to initial modifications on control flow.

3.0 <u>User Interface</u>

The success of any software technique is dependent upon its ease of use. The technique must be simple to understand and apply to the problem. This implies a high degree of automation in which the user interfaces with the technique at a very high level, and the technique is transparent to the user on how it operates.

The logical ripple effect analysis technique has been developed with these objectives. When the technique is fully automated, it is very easily applied to the problem. Although the technique is very sophisticated, the maintenance personnel applying the technique need only be concerned with its output. The logical ripple effect analysis technique is applied in the following three simple steps which are illustrated in Figure 4.

Step 1: Maintenance personnel utilize a change management system (CMS) to modify the program. The CMS consists of a text editor and a data base. The CMS records all of the changes in the program automatically in the data base. Thus, a record of the maintenance activity is created without special assistance from the maintenance personnel.

Step 2: After the modification of the program is complete, the maintenance personnel execute the lexical analysis package of the logical ripple effect analysis technique.

Step 3: Upon completion of the lexical analysis step, the maintenance personnel execute the tracing package of the logical ripple effect analysis technique. The tracing package utilizes the data base of program changes created by the CMS and maps these changes into the characterization of the program created by the lexical analysis step. It then traces logical ripple effect throughout the program. The output of the tracing package is a display of the code of the blocks affected by logical ripple effect. Another significant output of the tracing stage which is still undergoing research is the computation of a figure for the complexity of the proposed modification. This figure will reflect the amount of programmer's effort required to incorporate a particular program modification and take care of all its logical ripple effect. This figure can be used as a basis upon which various program modifications can be evaluated in terms of programmer's effort.

The logical ripple effect analysis technique, thus, provides maintenance personnel with valuable information about the maintenance activity without interfering with the maintenance process itself or requiring additional input from the maintenance personnel.

4.0 Outline of Logical Ripple Effect Analysis Technique

In this section, we will outline the processing steps involved with the lexical analysis and tracing phases of the logical ripple effect technique.

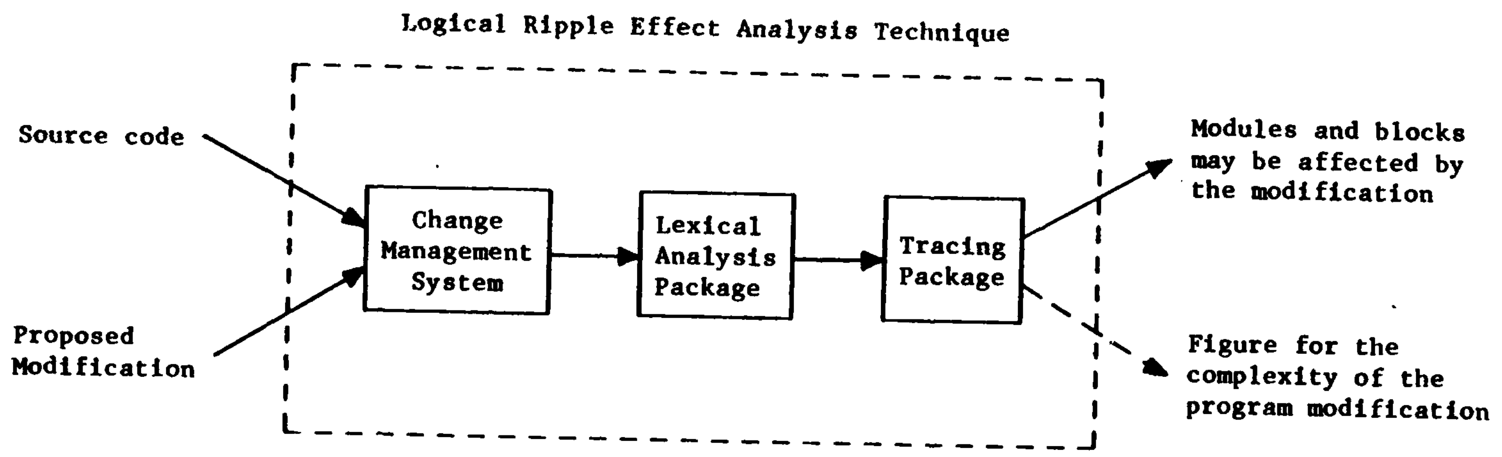
Logical Ripple Effect Analysis Technique
Source code
Proposed Modification
Change Management System
Lexical Analysis Package
Tracing Package
Modules and blocks may be affected by the modification
Figure for the complexity of the program modification

Figure 4. User interface level of the logical ripple effect analysis technique.

4.1 Lexical Analysis Phase

The first phase of the logical ripple effect analysis technique is the lexical analysis phase. In this phase, the program is analyzed with respect to the proposed modification and a characterization of the program is compiled and saved in a data base. The characterization of the program contains information necessary for tracing logical ripple effect. A description of the logical information needed for this characterization will now be presented.

4.1.1 Logical Characterization of the Program

The main purpose of the logical ripple effect analysis is to aid the maintenance personnel to better understand the scope of effect of his changes on the software system and identify program areas which must be checked to insure their logical consistency with the initial changes. To accomplish this, a modular large-scale software system is modeled by the set of modules and the interdependency between the modules. A program module is defined to be a separately invokable piece of the software system having single entry and single exit points. If a program module does not satisfy the single entry and single exit condition, it can easily be modified to satisfy this condition by using reference flags to refer to different entry points and different exit points. Thus, a module can correspond to a SUBROUTINE or PROCEDURE. The interdependencies between the modules can be represented by a directed graph, the invocation graph $\mathcal{R}$, where each node in $\mathcal{R}$ corresponds to one and only one module, and each edge denotes that the module corresponding to the tail of the edge invokes the module corresponding to the head of the edge at least once.

A program module is further partitioned into program blocks to reduce its complexity and enhance the information on the program structure.

A program block is a maximal sequence of computer statements having the property that each time when any statement in the sequence is executed, all are executed; except under the condition that the execution flow is transferred to another module, or information is input from or output onto a file, or a looping condition is defined in a DO type statement. When a module invocation is encountered, a sequence of three blocks may be assigned for this module invocation. An input statement would initiate a new block which contains the input statement. An output statement would terminate the current block which contains the output statement. A DO type statement which defines a looping condition constitutes a block itself. However, each program block has one single entry point and one single exit point.

A program module is modeled by the program graph associated with this module. A program graph is a directed graph where each node represents one and only one program block in the module and each edge represents the execution flow from the exit point of the tail node to the entry point of the head node.

In order to trace the logical ripple effect, it is necessary to characterize each block to reflect how potential errors can propagate within the block. A data usage is a data item which is referenced without change in an expression or part of an expression. A data definition is a data item whose value is modified in an expression or part of an expression. A control definition is an item assigned to a control directive to reflect the control condition, e.g. a control definition is assigned to an IF statement and the data items in the predicate are said to define the control definition. The potential propagator set for each block is defined to be the set of all usages in the block which can propagate potential errors. The source capable set for each block is defined to contain all definitions within this block which can cause potential errors to exist within this block. A flow mapping for each block is defined to associate each element in the potential propagator set with the elements in the source capable set such that potential errors can propagate from the potential propagator to the source capable definitions. The source capable set, the potential propagator set, and the flow mapping of a block together constitute the block's error characteristics.

Tracing of logical ripple effect also requires the characterization of each module to model the potential error behavior between the module and its surrounding environment. This characterization is referred to as the module's error characteristics. The error characteristics for a module is represented by the module level potential propagator set, the module level source capable set and the module level flow mapping. The elements in the module level potential propagator set can cause potential errors to propagate within this module. The elements in the module level source capable set represent potential errors which can flow from this module or remain within the module. The module level flow mapping represents how potential errors can propagate from the module level potential propagators to the module level source capable definitions.

A module must have its error characteristics defined after the error characteristics of the modules that are invokable by this module have been defined; otherwise, the error characteristics of the module cannot be completely specified. Thus, the order in which the module error characteristics are derived for all the modules in the software system is very important. A precedence ordering among the modules is defined to be a partial ordering such that if a module invokes another module, the former is assigned a higher precedence than that of the latter. Therefore, the error characteristics for all modules should be defined starting from the module with lowest precedence, i.e. a module that does not invoke any modules, and then proceeding to the modules with higher precedences according to the precedence ordering.

Thus, lexical analysis to produce a logical characterization of the program requires that for each module in the software system, the module's text is statically scanned to produce a program graph based on program blocks. The error flow properties of each program block represented by the potential propagator set, the source capable set and the flow mapping are also characterized. In addition, the module invocation graph is constructed to denote that a block in a module invokes another module. This process is dependent of the high level language in which the software system being analyzed is written. This process is referred to as Text-Level Lexical Analysis and different Text Analyzers have to be developed for different programming languages. A text is defined to be an entity that is compiled independently. As an example, a text may be a compool, a main program, or a subprogram in JOVIAL.

After all the texts in the software system have been processed by the Text Analyzer, the precedence ordering among modules will be derived. Then according to this precedence ordering, the module error characteristics for each module will be derived. At the same time, the error characteristic sets and flow mappings for those blocks which invoke some modules will be updated. This process if referred to as the System-Level Lexical Analysis and is independent of the programming language in which the software system is written.

4.1.2 Outline of the Procedure to Perform Lexical Analysis

The processing steps involved with lexical analysis can be summarized as follows:

Step 1: Perform Text-Level Lexical Analysis to produce a program graph based on program blocks, compute the error flow properties of each program block, and construct the invocation graph.

Step 2: Perform System-Level Lexical Analysis to derive the precedence ordering among modules, compute the module error characteristic sets, and update the block error characteristic sets.

4.2 Tracing Phase

The second phase of the logical ripple effect analysis technique consists of tracing the logical changes, i.e., the logical ripple effect which occurs as a consequence of the maintenance changes. The input to the technique in this phase includes all of the information about the program collected and stored in a data base during the lexical analysis phase.

4.2.1 Logical Ripple Effect Tracing

Tracing logical ripple effect is a very difficult problem and requires identification of error sources which will be utilized as starting points for the tracing. There are two types of error sources:

Primary error sources which are all the program definitions involved in the initial modification. Inconsistency of these program definitions can propagate from the primary error sources to other program areas.

Secondary error sources which are data or control definitions implicated through the usage of primary error sources and must be examined to insure that they are not inconsistent with the data items involved in the initial change.

The error sources that may flow from a block are represented by the propagation error source set for the block.

The algorithm to compute logical ripple effect operates upon each module characterization to trace error sources from their points of definition to their exit points. The algorithm is initialized with a set of modules and their primary error sources involved in the initial change. For each module initially involved in the modification, the algorithm traces the intramodule flow of potential errors from the primary error sources through the various program blocks. When the flow of error sources stabilizes, the algorithm applies a block identification criterion to determine which blocks within the module must be examined to insure that they are not logically inconsistent with the initial change. The block identification criterion is used to distinguish between blocks which are affected by the error flow and those which are not. A block is affected by error flow and, thus, may require further maintenance if the intersection of the block's propagation error source set and its source capable set is not empty. After the block identification is complete, a propagation criterion is applied to this module to define those error sources which flow from this module to those modules invoked by this module, and to modules which invoke this module. Error flow across module boundaries constitutes intermodule error flow. For each module affected by intermodule error flow, the algorithm traces intramodule error flow in the same manner as described above to determine the net effect that the propagated error sources have on their respective modules. The algorithm is executed in this manner until intermodule error flow stabilizes. An intermediate result obtained at the point is the set of modules which are affected by the intermodule error flow of error sources created by the primary error sources involved in the change. Then, a <u>logical ripple effect criterion</u> is applied to each module affected by intermodule error flow to determine if the module requires additional maintenance activity. The logical ripple effect criterion consists of examining the intersection of the propagation error source set and the source capable set for every block in the module. If the intersection is empty for each block in the module which is not specified for a module invocation, (i.e. the module is not affected by logical ripple effect because it does not cause inconsistency within itself), then the module requires no further maintenance activity. However, if there exists at least one block such that the intersection of the propagation error source set and the source capable set is not empty, then the module is affected by logical ripple effect.

A <u>block elimination criterion</u> is also applied to each module affected by intermodule error flow to identify the program blocks and their error sources which require additional maintenance. The block elimination criterion distinguishes between those blocks which are affected by the logical ripple effect and those which only contribute to the error flow. If a module is not affected by the logical ripple effect, then all blocks in the module and all blocks assigned for invocations to this module require no further maintenance activity since the error sources do not disturb this module's consistency.

4.2.2 Outline of the Procedure to Perform Ripple Effect Tracing

In this section, the processing steps for tracing logical ripple effect will be presented in the required order.

Step 1: Utilizing the change management system data base and the characterization of the program produced during lexical analysis, identify the set of blocks and their primary error sources initially involved in the change for each module in the program.

Step 2: Form a set $\bar{\mathcal{M}}$ composed of modules initially involved in the change.

Step 3: Compute the error flow of set $\bar{\mathcal{M}}$. Let the set of modules affected by the error flow be $\mathcal{M}^*$ and the set of blocks and their error sources within each module $\mathcal{M}_j$ which contributes to error flow be L_j.

Step 4: Apply the logical ripple effect criterion to each element in $\mathcal{M}^*$. Let all modules in $\mathcal{M}^*$ which require additional maintenance due to the logical ripple effect criterion form the set $\mathcal{M}^R$.

Step 5: Apply the block elimination criterion to each element in $\mathcal{M}^*$. Let all blocks and their error sources within M_j which require additional maintenance activity form the set L_j^R. The maintenance personnel must check all of the blocks in L_j^R for each module in $\mathcal{M}^R$ to insure that they are consistent with the initial change.

5.0 Description of Each Step of the Technique

In this section, a description of each of the steps involved in the lexical analysis and tracing phases of the logical ripple effect analysis will be provided. The description will be informal and concise. The processing steps will be described at a level which is language independent. Informal algorithms and approaches used in these steps will also be presented, but the actual implementation is language dependent and hence, omitted.

5.1 Description of Lexical Analysis Steps

In this section, a description will be presented for each of the lexical analysis steps outlined in Section 4. Section 5.2 will contain a description of each of the steps in the tracing phase which has also been outlined in Section 4.

5.1.1 Lexical Analysis Step 1

This step performs the Text-Level Lexical Analysis on each text to derive the program graphs, module invocation graph, block error characteristics, etc.

Since most programming languages allow several names to refer to the same memory location and also several memory locations referred by the same name, we have to resolve these address conflicts first in order to correctly trace the error flow within the software system. Therefore, this process is further decomposed into two passes.

5.1.1.1 Pass 1

The Pass 1 scans the text and performs the following main functions: to establish symbol tables and alias relations, to resolve address conflicts and to identify the global and passed parameter sets of modules defined within this text.

There are two types of address conflicts which must be resolved. One is called symbolic aliasing which arises when several names refer to the same memory location. The other is called the address aliasing which occurs when several memory locations may be referred by the same name.

Most programming languages permit the programmer to declare data items with the same name, but different scopes of effect. This capability can introduce address aliasing. One way to resolve this problem is to keep track of the scopes of effect for all data names by appropriate symbol tables or stacks, such as a common symbol table or program symbol stack. When a data name is referenced in the program, the data item with the scope of effect on this reference can be resolved from the symbol tables or stacks following the name resolution rules associated with the language. Then, the reference is relabelled to reflect which data item has the applicable scope of effect, e.g. by prefixing the name with a prefix which denotes the module where the data item having the scope of effect for the reference was declared.

Some programming languages allow the user to declare several data names for the same data item. The EQUIVALENCE statement in FORTRAN is an example. This capability can introduce one form of symbolic aliasing. This kind of symbolic aliases can be identified by seeking out the syntactic constructs used by each language to define the alias relation. Once identified, this kind of symbolic aliasing can be resolved by substituting only one element in the alias grouping for all other elements in the group throughout the scope of effect of the aliases.

Thus, in Pass 1, the symbol tables and alias relations must be established and used to resolve the address conflicts. In addition, the global and passed parameter sets of each module defined within the text can be identified as a by-product of the name resolution. The formal parameters, if there are any, appearing in the module declaration in their prefixed form are put into the passed parameter set of the module. The data items referenced within the module which have scopes of effect over the module are in the proper form which can reflect the respective scopes of effect of the data items and put into the global parameter set of the module.

A temporary file is written by Pass 1 as the card image of the text except that all aliases have been resolved and relabelled.

5.1.1.2 Pass 2

The Pass 2 scans the temporary file written by Pass 1 and performs the following main functions: to derive the program graphs for modules defined in the text, to identify blocks and their error characteristics, and to derive the module invocation graph. Note that Pass 2 can be skipped for texts which contain no executable statements and are used merely for declaration purpose, such as compools in JOVIAL.

For each programming language, a set of block segmentation conditions must be identified according to the syntactic construct of the language. These block segmentation conditions should identify all control flow changes other than those in sequential control flow, e.g. conditional statements, jumps, loops, etc., and some special conditions for error flow analysis, e.g. module invocation, data declaration, input and output statements. Associated with the block segmentation conditions are the predecessor-successor identification conditions which are used to build the predecessor-successor relationship among the blocks identified by the block segmentation conditions. As an example, when an unconditional GOTO statement which branches to a labelled statement is encountered, the block segmentation conditions should identify this and terminate the current block while the predecessor-successor identification conditions should specify that the current block is a predecessor of the block which contains the statement bearing the label referred by the GOTO statement. The block segmentation conditions and predecessor-successor identification conditions partition a module into program blocks and derive the program graph of the module in terms of the blocks and the predecessor-successor relationship among the blocks within the module.

In order to identify the error characteristics of blocks, the schemes to identify the usages and definitions from all types of statements, intrinsic functions and procedures must be identified according to the syntactic construct of the language. As an example, the simple data item appearing on the left-hand side of an assignment operator will be identified as a definition, and the data item(s) appearing on the right-hand side of the assignment operator will be identified as usage(s).

A block's error characteristics are represented by the potential propagator set, the source capable set, and the flow mapping of the block. A block error characteristics identification scheme has been developed to process the usages and definitions identified within a block to determine how they should be added to the potential propagator set and source capable set, respectively, and how the flow mapping should be constructed. A discussion of the scheme can be found in [12].

Pass 2 scans through the temporary file written by Pass 1. Once a new module scope is entered, an entry block with empty error charactistics will be specified for the module. When one or more of the block segmentation conditions are satisfied, the current block will be terminated and a new block is built with appropriate predecessor-successor relationship specified by the respective predecessor-successor identification conditions. The usages and definitions in a block are identified by the schemes from the statements, intrinsic functions and procedures appearing in the block. The usages and definitions are processed by the block error characteristics identification scheme to construct the potential propagator set, the source capable set, and the flow mapping of the block. At the end of the module scope, an exit block with empty error characteristics is specified for the module. The program graphs, the blocks and their respective error characteristics can thus be derived. Note that the information to indicate the entry points of blocks should be inserted and written with the temporary file which contains the text after resolving address conflicts on an output file. Thus, the output file can be scanned to identify the primary error sources, the blocks and modules involved in the initial changes.

When a module invocation is recognized by its syntactic construct or by name resolution, a sequence of three blocks may be assigned in the invoking module to establish the potential error flow between the invoking and the invoked modules. The first block in the sequence is used to construct the potential error flow between the actual input parameters and their corresponding formal input parameters. The second block in the sequence is referred to as a module invocation block and used to represent the potential error properties of the invoked module. The third block in the sequence is used to construct the potential error flow between the actual output parameters and their corresponding formal output parameters.

The two blocks used to construct the error flow between actual and formal parameters are required because the parameters passed between modules can introduce another form of symbolic aliasing. Furthermore, as far as the formal and actual parameters are concerned, the invoked module's error characteristics are expressed in terms of the formal parameters, while the local blocks' error characteristics in the invoking module are expressed in terms of the actual parameters. Thus, these two blocks are required to correctly trace the error flow between modules and preserve the invoking module's local block characteristics. Certainly, if the invoked module has no formal parameters, these two blocks can be omitted. Similarly, if the invoked module is a function, the block which is used to construct the error flow between output parameters can be omitted.

The block error characteristics of the blocks in the sequence assigned for a module invocation are initialized as empty in Pass 2 and will be specified later in the System-Level Lexical Analysis step after the invoked module's error characteristics have been defined. The information about the blocks assigned for the module invocation, the invoking module and the invoked module for each module invocation is stored in the module invocation table which will be used later in the System-Level Lexical Analysis step to update the block error characteristics of the blocks assigned for module invocations and in the tracing step to compute the logical ripple effect. Furthermore, the actual parameter list appeared in each module invocation is stored in conjunction with the module invocation.

For each module invocation, the invoking module is also specified as an immediate predecessor of the invoked module in the module invocation graph which will be used later in the System-Level Lexical Analysis step to derive the module precedence ordering and further in the tracing step to compute the logical ripple effect.

5.1.2 Lexical Analysis Step 2

At the beginning of this step the text-level lexical analysis has already produced a program graph based on program blocks, computed the error flow properties of each program block, and constructed the invocation graph. This step performs system-level lexical analysis to derive the precedence ordering among modules, compute the module error characteristic sets, and update the block error characteristic sets.

5.1.2.1 Derivation of Module Precedence Ordering

The module precedence ordering is used to determine the order in which the module error characteristics for all modules in the program are derived. It is a partial ordering and has a one-to-one correspondence with modules in the program. If a module invokes another module, then the former should be assigned a higher precedence than that of the latter.

The module precedence ordering can be derived from the module invocation graph which was created in the Text-Level Lexical Analysis stage. Recall that the module invocation graph is completely characterized by a set, where each element in the set consists of a module and the set of modules invoked by this module. The module precedence ordering is represented by a set where each element consists of a module and its precedence.

The module precedence ordering can be derived by the following algorithm:

<u>Step 1</u>: Initialization

Construct a module list to contain all the modules in the program. Initialize the module precedence ordering to be an empty set. Initialize a temporary module list to be empty. Finally, set the precedence counter to be zero.

<u>Step 2</u>: Ordering and Selection

For each module in the module list, search the module invocation graph to determine if the set of immediate successors of this module is empty. If it is empty, then increment the precedence counter, add the module and the current value of the precedence counter into the module precedence ordering set, delete this module from the module list, and finally add the module into the temporary module list.

Step 3: Termination

If the module list is empty, i.e. all the modules in the program have been processed, then terminate the program.

Step 4: Selection

Select a module from the temporary module list and then delete it from the list.

Step 5: Deletion

For each module contained in the module list, search the set of immediate successors of the module to determine if the set contains the module selected from the temporary module list. If it does, then delete the module selected in Step 4 from the set of immediate successors of the module.

Step 6: Repetition

If the temporary module list becomes empty (i.e. all the modules which were assigned precedences in Step 2 have been deleted from the sets of immediate successors of the modules which invoke these modules), then go to Step 2 to process the modules whose immediate successors have been assigned precedences; otherwise, go to Step 4 to process the next module in the temporary module list.

5.1.2.2 Derivation of Error Characteristics for a Module

The module's error characteristics are used to model the potential error behavior between the module and its surrounding environment. The module's error characteristics are represented by the module level potential propagator set, the module level source capable set, and the module level flow mapping. The elements in the module level potential propagator set can cause potential errors to propagate within this module. The elements in the module level source capable set represent potential errors which can flow from this module or remain within the module. The module level flow mapping represents how potential errors can propagate from the module level potential propagators to the module level source capable definitions.

The module can only interface with its surrounding environment via its parameter list. The elements in a module's parameter list may have passed attributes or global attributes. The passed and global parameters were identified and stored in the passed parameter set and global parameter set, respectively, for the module during the Text-Level Lexical Analysis.

The module's error characteristics can be algorithmically derived in 8 steps:

Step 1: Augment the global parameter set of the module.

The global parameter set of the module is augmented to contain all augmented global parameter sets of the modules which are immediate successors of this module. Because a global variable may not be used in the module but may be used in a module invoked by the module, the global variable must be added to the parameter list of the module to preserve the local error characteristics of the module. The parameter list of the module is augmented too by taking the union of passed parameter set and the augmented global parameter set of the module.

Step 2: Calculate the set of natural source capable definitions.

The elements in the set of natural source capable definitions represent the potential error sources which can flow from this module back to an invoking module. The intersection of the module's parameter list and the union of all source capable sets for the blocks in the module defines the natural source capable set.

Step 3: Calculate the potential propagator candidate set.

The elements in the potential propagator candidate set represent the elements in the module's parameter list which are suspected of propagating potential errors into the module. The intersection of the module's parameter list and the union of all potential propagator sets for the blocks in the module defines the potential propagator candidate set.

Step 4: Calculate the natural potential propagator set and the pseudo potential propagator set.

The elements in the natural potential propagator set represent the elements in the module's parameter list which can cause potential errors to exist within the module and flow out of the module. The elements in the pseudo potential propagator set represent the elements in the module's parameter list which can cause potential errors to exist and remain within the module. A potential propagator identification function is used to examine each element in the potential propagator candidate set to determine if it can propagate potential errors into the module. The potential propagator identification function treats the potential propagator candidate as a primary error source and makes it the only member of the propagation error source set for the entry node of the module. Then, the potential propagator identification function traces the error flow within the module using the intramodule error flow algorithm. Finally, the propagation error source set for the exit node of the module is examined. Suppose that the set is not empty, i.e. the potential propagator candidate can propagate potential errors to the elements in the set. Then, the intersection of the natural source capable set and the propagation error source set for the exit block of the module is examined. If the intersection is not empty, the potential propagator candidate can propagate potential errors to the elements in the intersection which can flow out of the module and hence the potential propagator candidate is added into the natural potential propagator set. Otherwise, i.e. the intersection is empty, all the error sources created by the potential propagator candidate remain within the module and hence the potential propagator candidate is added into the pseudo potential propagator set.

Step 5: Calculate the module level potential propagator set.

The module level potential propagator set is defined as the set of parameters which can cause potential errors to exist within the module. The union of the natural potential propagator set and the pseudo potential propagator set defines the module level potential propagator set.

Step 6: Calculate the pseudo source capable set.

A unique pseudo source capable definition is defined for each element in the pseudo potential propagator set to represent the error sources which are created by the pseudo potential propagator and remain within the module. The existence of the pseudo source capable set is required for preservation of the invoking module's local block error characteristics as expressed by the source capable set and propagation error source set for the local block. Preservation of these error characteristics insures that intramodule error flow algorithm will correctly identify those local blocks which are affected by a modification and invoke this module. Elements in the pseudo source capable set can be arbitrarily defined in such a manner that they will not create any erroneous secondary error sources in the invoking module.

Step 7: Calculate the module level source capable set.

The module level source capable set consists of elements in the natural source capable set and the elements in the pseudo source capable set.

Step 8: Identify the module level flow mapping.

The module level flow mapping maps each element in the module level potential propagator set to the elements in the module level source capable set which may be affected by the potential propagator. The elements in the natural potential propagator set are mapped to the respective elements in the intersection of the natural source capable set and the propagation error source set for the exit node of the module as identified by the potential propagator identification function. The elements in the pseudo potential propagator set are mapped to the respective pseudo source capable definitions identified in Step 6.

Note that the module error characteristics for all modules in the software system should be derived according to the module precedence ordering identified in the previous section and starting from the module which has the lowest precedence.

5.1.2.3 Update of Block Error Characteristics

Recall that in Pass 2 of the Text-Level Lexical Analysis step, a sequence of blocks is assigned for each module invocation and the error characteristic sets for the blocks in the sequence are specified to be empty. Now, it is necessary to update the block error characteristic sets for these blocks. For each module invocation, the block error characteristic sets for the module invocation block in the invoking module are updated with the respective module error characteristics of the invoked module. That is, we assign the invoked module's module level potential propagator set, module level source capable set and the module level flow mapping to the potential propagator set, source capable set and flow mapping, respectively, for the module invocation block in the invoking module. This can easily be done in an implementation, by changing the pointers linking the blocks to the block error characteristics. The block error characteristic sets for the block which is used to construct the potential error flow between input parameters can be specified by treating the input actual parameters as potential propagators which can affect the corresponding input formal parameters. Similarly, the block error characteristic sets for the block which is used to construct the potential error flow between output parameters can be specified by treating the output formal parameters as potential propagators which can affect the corresponding output actual parameters. A formal parameter is identified to be an input formal parameter if it is an element in the module level potential propagator set. A formal parameter is identified to be an output parameter if it is an element in the natural source capable set for the module. The correspondence between formal and actual parameters can be seeked out by examining the formal parameter list of the invoked module and the actual parameter list which was stored in conjunction with this module invocation during the Pass 2 of the Text-Level Lexical Analysis step.

5.2 Description of Tracing Steps of Logical Ripple Effect Analysis

In this section, a description will be presented for each of the tracing steps of the logical ripple effect analysis outlined in Section 4.

5.2.1 Tracing Step 1 of Logical Ripple Effect Analysis

In this step, the set of blocks and their primary error sources involved in the change is identified for each module in the program.

A primary error source is defined to be a data or control definition which is directly affected or implicated by the initial modification. A directly affected primary error source is a definition whose value or control condition associated with it was directly changed by the initial modification. Implicated primary error sources are required because our technique starts tracing the logical ripple effect from the immediate successor blocks of the blocks which are involved in the initial modification and hence the maintenance programmer has to identify the definitions affected by the intra-block error flow within the primary error sources blocks. A definition in a primary error sources block is implicated as a primary error source if it is defined by direct or indirect usages of some directly affected primary error sources of the block. Note that, after a definition was identified as a primary error source, if it is redefined in the block without usages of any affected data items, then the definition can no longer propagate potential errors to other blocks, and hence should be removed from the set of primary error sources of the block. From now on, we will assume that the implication process is always carried out by the maintenance personnel.

Another type of complication arises when the control flow is changed due to deletion of code. Since our technique is based on the potential error properties of the modified program, some potential errors may not be traceable due to the change in control flow. In order to solve this problem, the maintenance personnel has to specify the deleted definitions as directly affected primary error sources for the blocks in the modified program, in which the deleted code could transfer execution flow. These blocks of the modified program should be specified as primary error sources blocks. In the following discussion, it is also assumed that this type of complication is always resolved by the maintenance personnel in case of deletion of code.

Based on the above discussion, our main emphasis here will be on how to identify the directly affected primary error sources due to a program modification. Note that a change, insertion, or deletion of a module invocation requires special care. Let us consider the following modifications:

* Suppose that the data items used to define a control condition were changed, e.g. a loop termination condition was modified. Then, the control definition associated with this control condition is specified as a directly affected primary error source of the block, where the control definition is assigned.
* Suppose that a data definition was changed, added, or deleted in a block. Then, the definition is specified as a directly affected primary error source of the block.
* Suppose that an actual parameter x was replaced by y in a module invocation. If the corresponding formal parameter f is an input parameter, then f is specified as a primary error source of the input parameter mappings block associated with this module invocation. If f is an output parameter, then x and y both are specified as primary error sources of the output parameter mappings block.
* Suppose that a module invocation which invokes a newly added or an existing module was inserted into the program. Then, the invoked module's natural source capable definitions are specified as primary

error sources of the module invocation block associated with this newly added module invocation.

* Suppose that a module invocation which invokes M_k was deleted from a module M_j. Then, the directly affected primary error sources are M_k's natural source capable definitions, except that the formal output parameters should be replaced by their corresponding actual parameters which appeared in the deleted module invocation.

5.2.2 Tracing Step 2 of Logical Ripple Effect Analysis

In this step, the set $\bar{\mathcal{M}}$ composed of modules initially involved in the change can be formed directly from the result obtained in the last step (Tracing Step 1). That is, if a module contains at least one primary error sources block in it, then the module is added into $\bar{\mathcal{M}}$.

5.2.3 Tracing Step 3 of Logical Ripple Effect Analysis

In this step, the error flow within the program is traced from the points of definition to the exit points of the error sources. A tracing algorithm, using the modules involved in the initial changes and identified in the last step as a starting point, operates upon each module characterization to trace the error flow. The intramodule and intermodule error flow models form the basis of this tracing step.

5.2.3.1 Intramodule Error Flow

Intramodule error flow emulates the error flow between blocks in a module.

The error sources which flow out of a block are represented by the propagation error source set of the block. From the propagation error source set of an immediate precedessor block, a tracing function is used to emulate the error sources which may flow out of the block as a result of the incoming error sources propagated from the immediate predecessor block. Obviously, the primary error sources identified in the block can flow out of the block. The incoming error sources may implicate new secondary error sources in the block or they may pass through the block. An incoming error source which is also a potential propagator of the block can propagate errors to the elements in the source capable set which are mapped by the potential propagator under the flow mapping, i.e. to the source capable definitions which are defined in the block by direct or indirect usages of the potential propagator. Thus, the new secondary error sources implicated by the incoming error sources can be identified as elements in the source capable set which are mapped by the elements in the intersection of the potential propagator set of the block and the propagation error source set of the immediate predecessor block. An incoming error source which is not redefined in the block can pass through the block. Hence, the set of incoming error sources which just pass through the block can be obtained by eliminating such incoming error sources which are also members of the source capable set of the block, i.e. by deleting the intersection of the source capable set of the block and the propagation error source set of the immediate predecessor block from the propagation error source set of the immediate predecessor block. Therefore, the tracing function emulates the error sources which may flow out a block as the union of the primary error sources identified in the block, the implicated new secondary error sources and the incoming error sources propagated from the immediate predecessor block and just passing through the block. The propagation error source set of a block can be derived by the union of error sources obtained by applications of the tracing function on the block for all immediate predecessors of the block.

An algorithm, called the <u>intramodule error flow algorithm</u>, is used to trace how the errors flow from the primary error sources blocks to the blocks in the module. It applies the tracing function on a block-immediate successor basis to propagate errors from the initial error sources blocks to all immediate successor blocks v_i of S, and then to all immediate successor blocks of v_i, etc. The tracing function is applied in this manner as long as new secondary error sources are created. When the flow of error sources stabilizes, the algorithm applies a block identification criterion to determine which blocks within the module are affected by the creation and propagation of secondary error sources.

The intramodule error flow algorithm can be stated as follows:

<u>Step 1</u>: Initialization

Initialize all propagation error source sets of the blocks in the module to containing no errors. Define a list which contains primary error sources blocks and initialize the propagation error source sets of these blocks to containing their respective primary error sources.

Step 2: Branch

If the list becomes empty, i.e. the error flow within the module has stabilized, then go to Step 6 to perform block identification.

Step 3: Selection

Select a block from the list and then delete it from the list.

Step 4: Propagation, Comparison and Update

For each immediate successor of the block selected in Step 3, apply the tracing function on it to produce the set of error sources which currently flows out of it. The set of error sources is then examined to see if it is contained in the current propagation error source set of the immediate successor block. If it is not, i.e. new secondary error sources have been implicated by the error sources which currently flow out of the selected block, then the immediate successor block is added into the list and the propagation error source set of the immediate successor block is updated by the union of the current propagation error source set and the set of error sources identified by the application of the tracing function.

Step 5: Repetition

Go to Step 2 to process all immediate successor blocks of a block which will be selected from the list.

Step 6: Block Identification

The flow of error sources within the module has stabilized. Thus, the block identification criterion is applied to each block in the module to see if the block is affected by the creation or propagation of the error sources. For each block, the intersection of the propagation error source set and the source capable set is examined. If the intersection is empty, then the block is not affected by the error flow because it is incapable of internally generating any secondary error source. Otherwise, the block is affected by the error flow and the elements in the intersection represent the definitions in the block which are affected by the error flow.

Step 7: Termination

Halt the program.

5.2.3.2 Intermodule Error Flow

The intermodule error flow emulates the flow of error sources across module boundaries of the software system.

Two a priori conditions must exist before intermodule error flow can occur: 1) there exist error sources which have the capability to propagate between two modules, 2) there exists an enabled path for error sources to propagate between two modules.

Error sources use communication paths (e.g. passed parameter list, shared data) and these paths are enabled at the time of a module's invocation. The error sources can flow in both directions between two modules. The error flow from the invoking module to the invoked module is called downward intermodule error flow, while the error flow from the invoked module back to the invoking module is called upward intermodule error flow.

A downward intermodule error flow criterion is used to check if the invoking module can propagate error sources to the invoked module. The intersection of the propagation error source set of the module invocation block in the invoking module and the invoked module's module level source capable set is examined. If the intersection is not empty, i.e. there exist error sources resulting from the intramodule error flow in the invoked module, then the invoked module is affected by the downward intermodule error flow and the error sources which propagate from the invoking module to the invoked module are referred to as the downward primary error sources of the invoked module. The downward primary error sources of the invoked module can be identified by taking the intersection of the invoked module's module level potential propagator set and the union of error sources which flow out of the immediate predecessor blocks of the module invocation block in the invoking module. The downward primary error sources will be added to the propagation error source set of the entry block in the invoked module, and used to identify secondary error sources that flow within the invoked module.

The upward intermodule error flow criterion is used to check if the invoked module can propagate error sources to the invoking module. The intersection of the propagation error source set of the exit block in the invoked module and the invoked module's natural source capable set is examined. If the intersection is not empty, i.e. error sources can flow from the invoked module as a direct result of the intramodule error flow in the invoked module, then the invoking module is affected by the upward intermodule error flow and the intersection defines the upward primary error sources of the invoking module. The upward primary error sources will be added to the propagation error source set of the module invocation block in the invoking module, and used to identify secondary error sources that flow within the invoking module.

Note that the overall flow of error sources throughout the program cannot be identified without the knowledge of both upward and downward error flows. The upward error flow cannot be identified without the knowledge of downward error flow. Thus, the downward error flow from a module to the modules invoked by the module must be traced before the upward error flow from the module to the modules which invoke the module can be traced.

An algorithm, called the intermodule error flow algorithm, is used to trace the error flow within the software system. This algorithm can be informally stated as follows:

Step 1: Initialization

Form a set, $\mathcal{M}^{\&}$, to be equal to the set $\overline{\mathcal{M}}$ which was derived in Tracing Step 2. The set $\mathcal{M}^{\&}$ is used to record the set of modules potentially affected by upward intermodule error flow. Initialize another set, $\mathcal{M}^{*}$, to be an empty set. The set $\mathcal{M}^{*}$ is used to contain the modules affected by intermodule error flow. For each module M_j in the program, the propagation error source set of the entry block and the set L_j are initialized to be empty, where L_j consists of the blocks in M_j affected by error flow and their associated error sources.

Step 2: Intermodule Error Flow Termination

If $\mathcal{M}^{\&}$ becomes empty, i.e. the intermodule error flow has stabilized, then terminate. Now, the set $\mathcal{M}^{*}$ contains all the modules affected by error flow, and the sets L_j's of the modules in $\mathcal{M}^{*}$ contain the blocks affected by error flow and their associated error sources. These sets will be used later to compute the logical ripple effect.

Step 3: Termination of Downward Error Flow Calculation

If $\overline{\mathcal{M}}$ becomes empty, i.e. the modules which influence intermodule error flow have all been processed, then go to Step 7 to identify the upward intermodule error flow on the modules in $\mathcal{M}^{*}$.

Step 4: Module Selection

Select a module from $\overline{\mathcal{M}}$ and then delete it from $\overline{\mathcal{M}}$. Let M_j denote the selected module.

<u>Step 5</u>: Intramodule Error Flow Tracing

Initialize a list to contain the primary error sources blocks in M_j. For each block in the list, initialize its propagation error source set to contain the primary error sources in the block. Here the primary error sources may be the error sources identified from the initial modification, the upward primary error sources or the downward primary error sources. For the blocks in the module which are not primary error sources blocks, initialize their propagation error source set to be empty.

Apply the tracing function on a block-immediate successor block basis to trace the intramodule error flow within M_j, as described in the Intramodule Error Flow Algorithm.

When the intramodule error flow in M_j stabilizes, apply the block identification criterion to blocks in M_j and add the blocks affected by intramodule error flow and their associated error sources into L_j.

<u>Step 6</u>: Application of Downward Intermodule Error Flow Criterion

For each block contained in L_j, search the module invocation table to see if the block is a module invocation block. If it is, calculate the set of error sources which currently flow into the module invocation block from its immediate predecessor blocks by taking the union of the propagation error source sets of the immediate predecessor blocks. Then, check if the intersection of that set and the module level potential propagator set of the invoked module properly contains the propagation error source set of the entry block in the invoked module. If it does, i.e. new error sources flow into the invoked module, then add the invoked module into $\mathcal{M}^*$ and $\bar{\mathcal{M}}$. Furthermore, the entry block is added into the set of primary error sources blocks of the invoked module, while the primary error sources set of the entry block is updated by the union of the current propagation error source set of the entry block and the intersection derived above.

After all blocks in L_j have been examined, go to Step 3 to continue calculating the net effect on the modules in $\mathcal{M}^\&$ and the modules invoked by these modules.

<u>Step 7</u>: Application of Upward Intermodule Error Flow Criterion

For each module in $\mathcal{M}^\&$, apply the upward intermodule error flow criterion. Let M_j be a member of $\mathcal{M}^\&$. Calculate the intersection of the propagation error source set of the exit block in M_j and the natural source capable set of M_j. This intersection defines the upward primary error sources of M_j. If the intersection is empty, i.e. no error sources currently flow from M_j, then examine another module in $\mathcal{M}^{\&}$. Otherwise, search the module invocation table and add the modules which invoke M_j into $\bar{\mathcal{M}}$ and $\mathcal{M}^*$. Furthermore, for each module which invokes M_j, add the blocks in the module which invoke M_j into the set of primary error sources blocks of the module and update the sets of primary error sources of the blocks by the upward primary error sources of M_j.

After all modules in $\mathcal{M}^{\&}$ have been examined, assign $\bar{\mathcal{M}}$ to $\mathcal{M}^{\&}$ and go to Step 2 to identify the net effect on modules currently in $\mathcal{M}^{\&}$ and the modules invoked by members in $\mathcal{M}^{\&}$.

5.2.4 <u>Tracing Step 4 of Logical Ripple Effect Analysis</u>

In this step, the set of modules which are affected by logical ripple effect, as denoted by $\mathcal{M}^R$, is identified from $\mathcal{M}^*$, which is the set of modules affected by the error flow. Recall that $\mathcal{M}^*$ was derived previously in Tracing Step 3.

A module affected by error flow may not contribute to the logical ripple effect if the error sources only pass through the module without disturbing the consistency of the module. A logical ripple effect criterion is used to check if a module is not only affected by error flow, but also by logical ripple effect. For a module in $\mathcal{M}^*$, if the intersection of the propagation error source set and the source capable set is empty for every block in the module which is not assigned for a module invocation, then the module is not affected by logical ripple effect because all error sources only pass through the module to the modules invoked by the module. Otherwise, at least one definition in the module is affected by error flow and the module is affected by logical ripple effect and hence requires further maintenance activity.

In this step, $\mathcal{M}^R$ is first initialized to be empty. Then, the logical ripple effect criterion is applied to each module in $\mathcal{M}^*$. If a module is identified as affected by logical ripple effect, then it is added into $\mathcal{M}^R$. After all the modules in $\mathcal{M}^*$ have been examined, the set $\mathcal{M}^R$ contains the set of modules affected by logical ripple effect.

5.2.5 Tracing Step 5 of Logical Ripple Effect Analysis

In this step, the set of blocks and their error sources which are affected by logical ripple effect is identified from the set of blocks and their error sources which are affected by error flow. Recall that for each module M_j in $\mathcal{M}^*$, the set of blocks in M_j and their error sources affected by error flow was derived in Tracing Step 3 and denoted by L_j.

A block affected by error flow may not be affected by logical ripple effect. A block elimination criterion is used to eliminate the blocks which are not affected by logical ripple effect from the set of blocks which are affected by error flow. If a module M_j in $\mathcal{M}^*$ was identified in the last step as not affected by logical ripple effect, then all blocks in M_j require no further maintenance activity because M_j's consistency was not disturbed. Furthermore, the blocks which are assigned in other modules for invocations to M_j are not affected by the logical ripple effect for the same reason.

Hence, the set of modules which are only affected by error flow will be formed first by taking the set difference of $\mathcal{M}^*$ and $(\mathcal{M}^* \cap \mathcal{M}^R)$, where $\mathcal{M}^* \cap \mathcal{M}^R$ is the set of modules affected by both error flow and logical ripple effect. For each module M_j in $(\mathcal{M}^* - (\mathcal{M}^* \cap \mathcal{M}^R))$, the set L_j^R is specified to be empty. The set L_k^R is assigned the set L_k for each module M_k in $\mathcal{M}^R$. Then, for each module M_j in $(\mathcal{M}^* - (\mathcal{M}^* \cap \mathcal{M}^R))$, delete the blocks and their error sources which are assigned for invocations to M_j from the respective L_k^R's for which M_k's invoke M_j.

After all modules affected only by error flow have been processed, the L_j^R's will contain the blocks affected by logical ripple effect and their associated error sources. The maintenance personnel should check the blocks and their error sources in the L_j^R's to insure their logical consistency with the initial modification.

6.0 References

[1] Yau, S. S., Collofello, J. S., and MacGregor, T., "Ripple Effect Analysis of Software Maintenance," Proc. of COMPSAC 78, pp. 60-65.

[2] Rye, P., Bamberger, F., Ostanek, W., Brodeur, N. and Goode, J., Software Systems Development: A CSDL Project History, RADC-TR-77-213, pp. 33-41.

[3] Goodenough, J. B., and Zara, R. V., "The Effect of Software Structure on Software Reliability, Modifiability, and Resuability: A Case Study and Analysis," Softech Incorporated, July 1974, p. 82.

[4] McCall, J. A., Richards, P. K., and Walters, G. F., Factors in Software Quality, Volumes I, II, and III, General Electric Company, pp. 2-3, 3-5, 7-9.

[5] Goullon, H., Isle, R., and Lohr, K., "Dynamic Restructuring in an Experimental Operating System," Proc. Third International Conf. on Software Engineering, 1978, pp. 295-304.

[6] Ringland, G. and Trice, A. R., "Pilot Implementations of Reliable Systems," Software Practice and Experience, Vol. 8, May-June 1978, pp. 323-339.

[7] Yourdon, E. and Constantine, L., Structured Design, Yourdon, Inc., 1976, p. 392.

[8] Boehm, B. W., "Software and Its Impact: A Quantitative Assessment," Datamation, May 1973, pp. 48-59.

[9] Zelkowitz, M. V., "Perspectives on Software Engineering," ACM Computing Surveys, Vol. 10, No. 2, June 1978, pp. 197-216.

[10] Ramamoorthy, C. V., and Ho, S. F., "Testing Large Software with Automated Software Evaluation Systems," Current Trends in Programming Methodology, Volume II, (R. Yeh, ed.), Prentice-Hall, Inc., 1977, pp. 112-150.

[11] Haney, F. M., "Module Connection Analysis--A Tool for Scheduling Software Debugging Activities," Proc. Fall Joint Computer Conf., 1972, pp. 173-179.

[12] Yau, S. S., "Self-Metric Software--Summary of Technical Progress," Vol I Final Technical Report.

[13] Yau, S. S. and Collofello, J. S., "Self-Metric Software, Vol III"--A Handbook: Part II, Performance Ripple Effect Analysis", Final Technical Report.

[14] Yau, S. S. and Collofello, J. S., "Performance Considerations in the Maintenance Phase of Large-Scale Systems," RADC-TR-79-129, June 1979.

[15] Yau, S. S. and Collofello, J.S., "Performance Ripple Effect Analysis for Large-Scale Software Maintenance," RADC-TR-80-55, December 1979.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

DAT
FILM